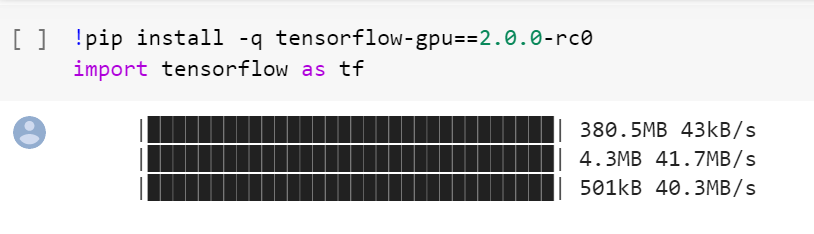
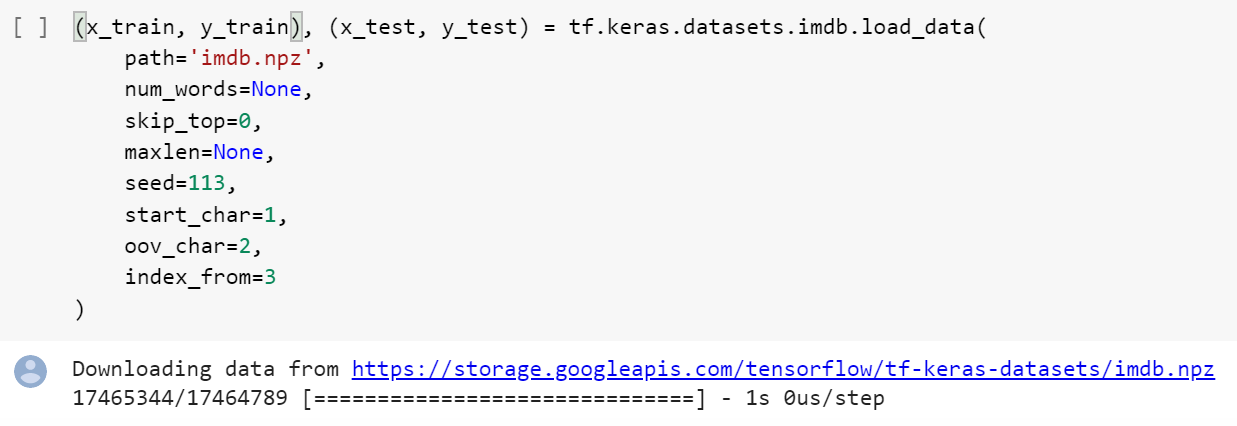
Solution for text generation and classification

Install Tensorflow



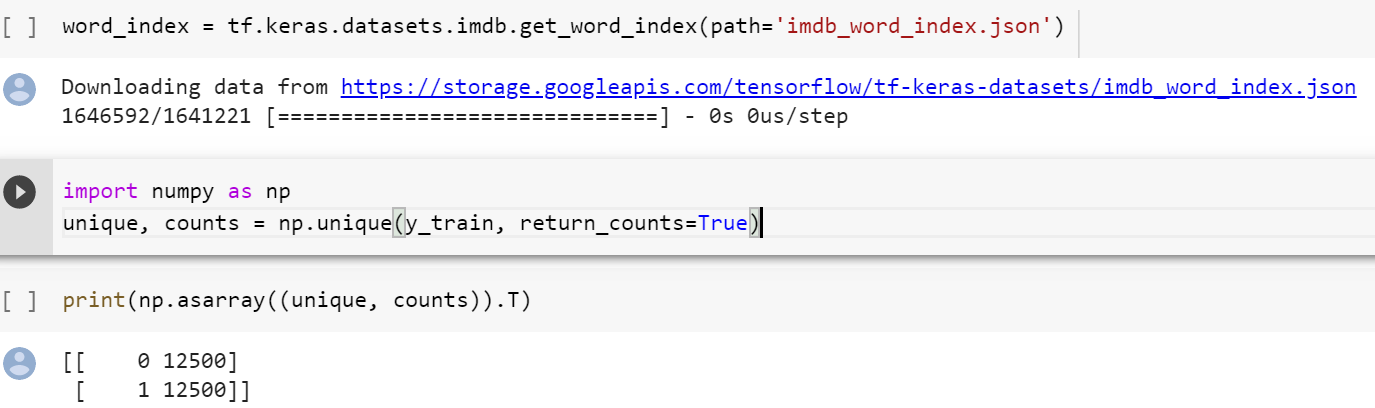
## **IMDB data**

Let's load the IMDB movie review data. This is already preprocessed and is a part of tf.keras.datasets for your convenience. This data is encoded in numpy array, and we would need its dictionary to decipher it to plain text.

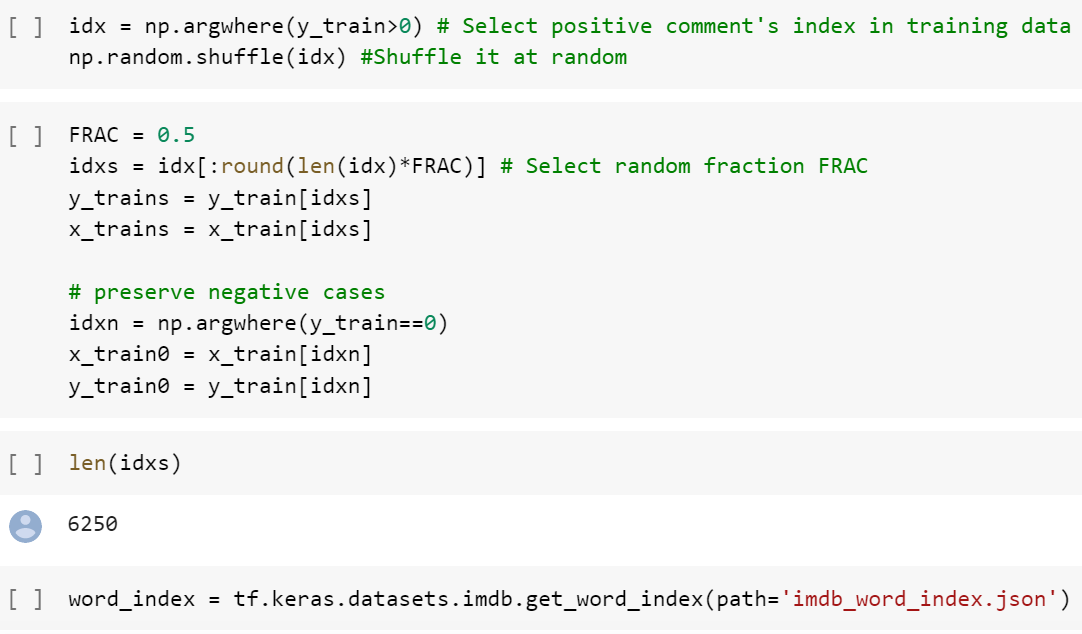


### Dictionary (tokens)

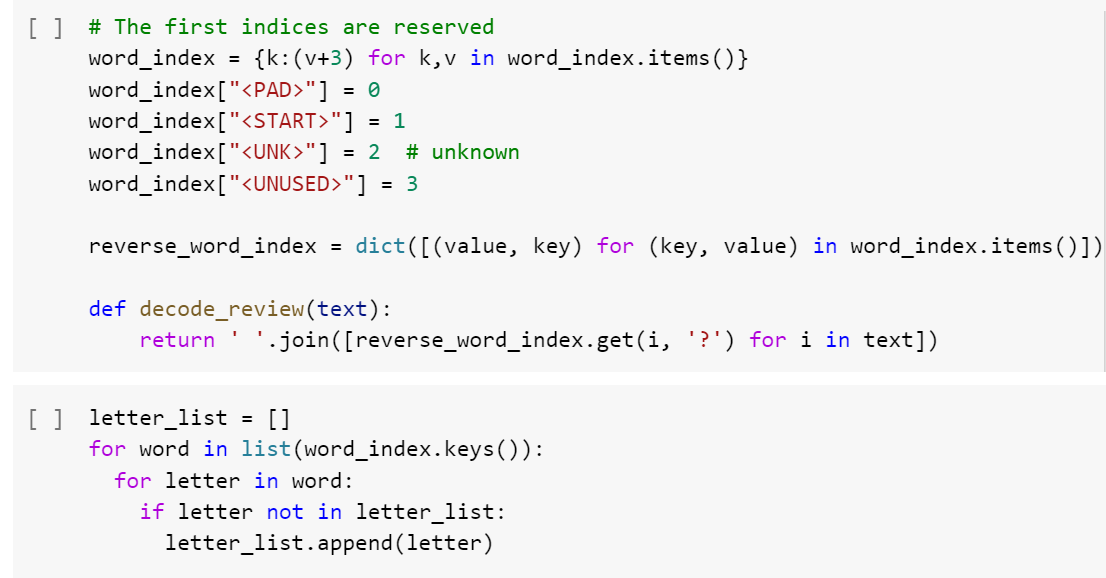
Let's also take advantage of the dictionary included in this dataset by reusing it.



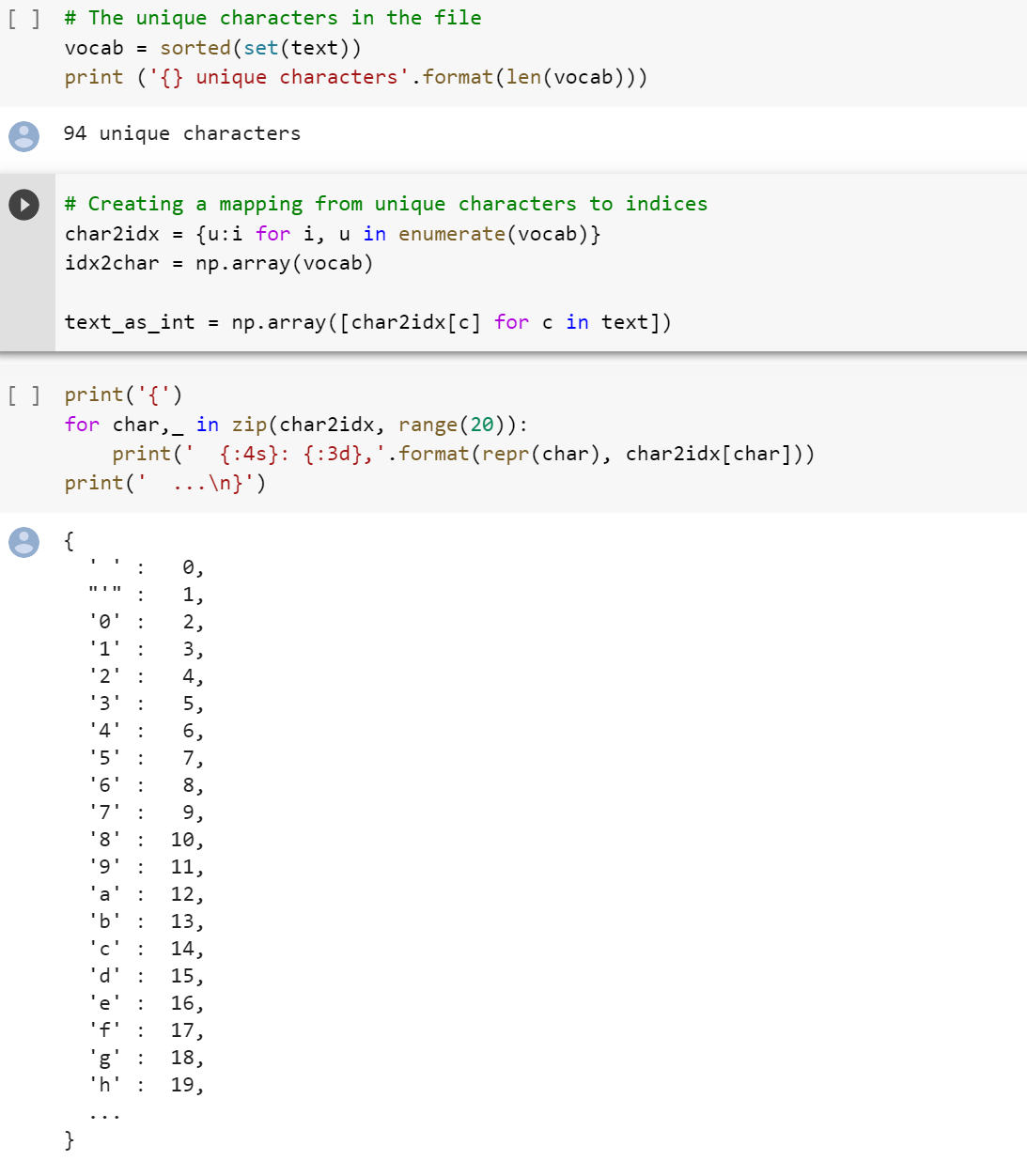
The data contains 12,500 positive and negative reviews as indicated by label counts. Let's arbitrarily discard half of positive reviews. This renders our training data to be imbalanced for our purpose. We want to generate brand new reviews to make the training data balanced again. Our hypothesis is that brand new reviews can be used to augment imbalanced training data.

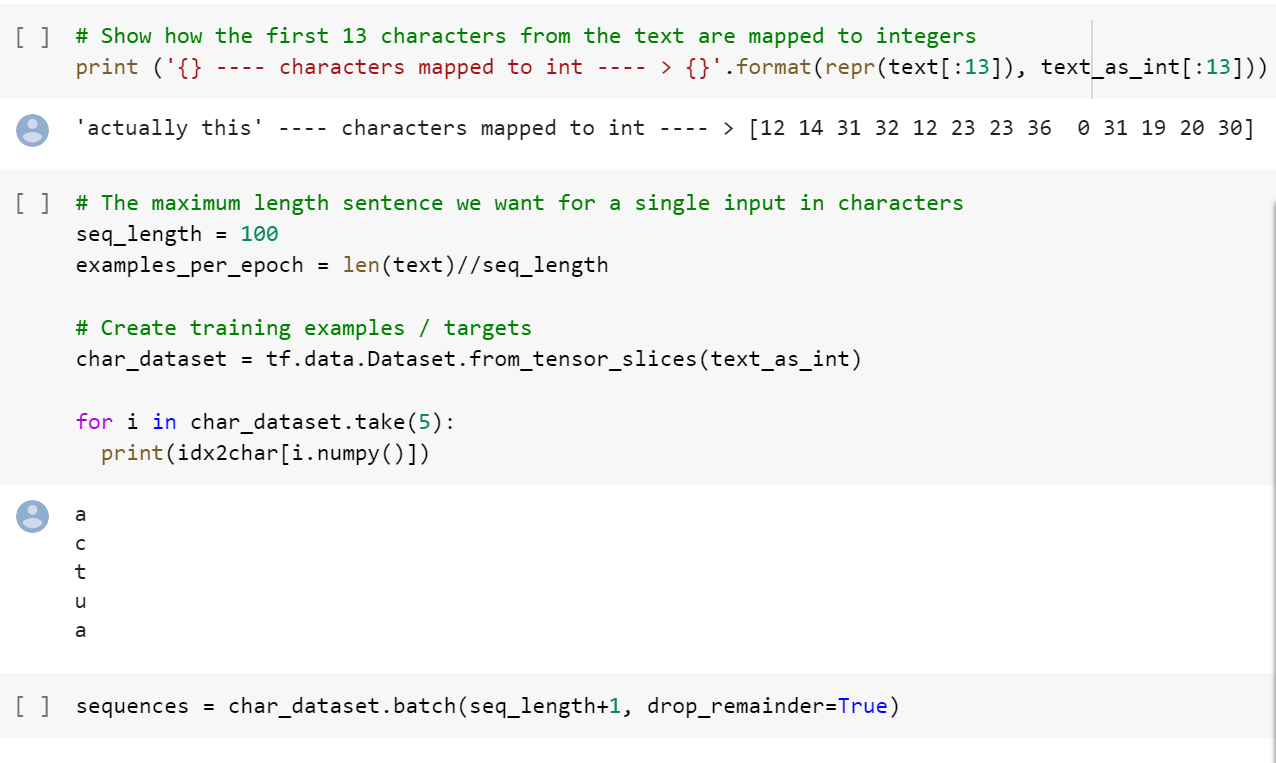


Let's append to the dictionary some extra tokens to handle padding for short reviews, as well as tagging start of a review and designate tokens to handle exceptions for unknown or unused words.

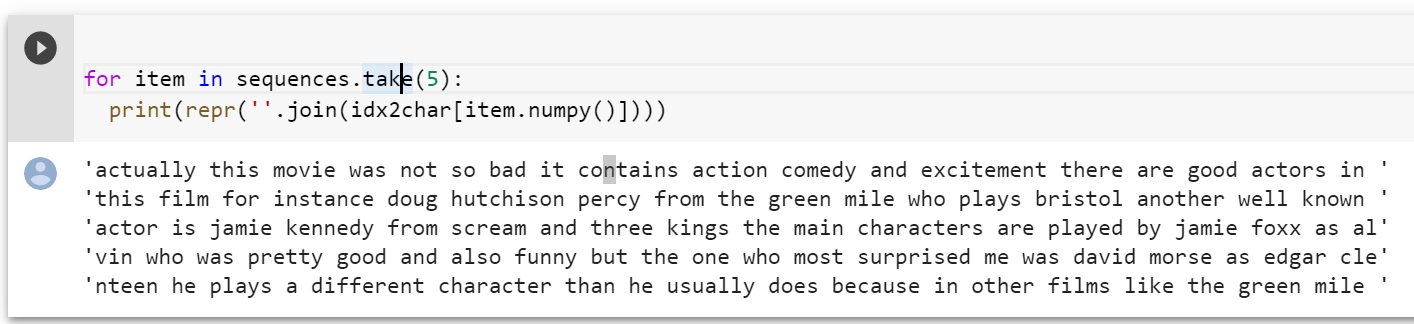






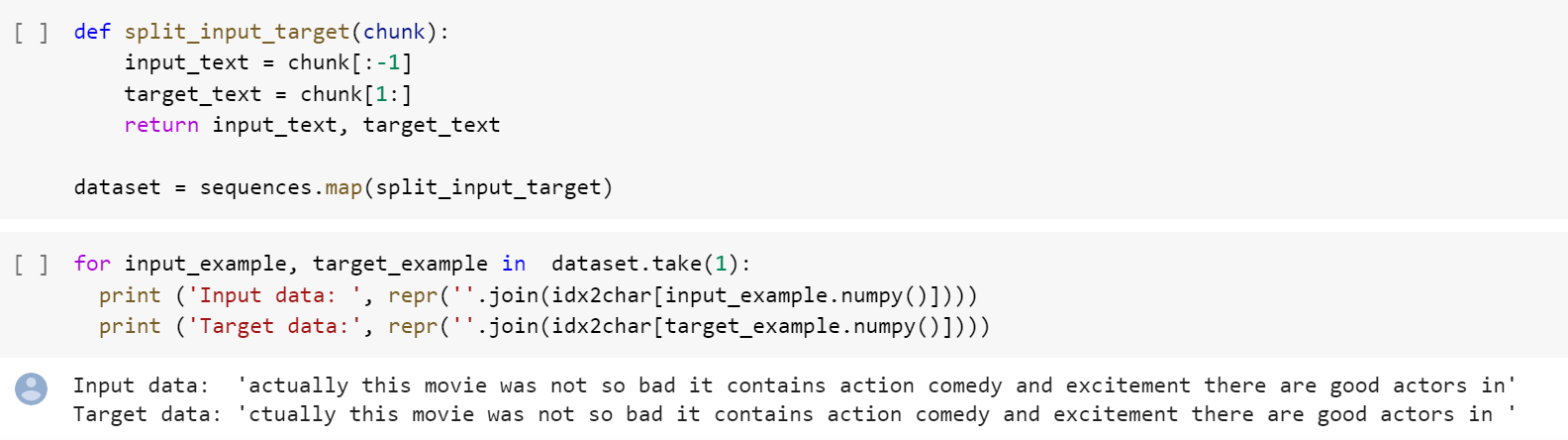


Let's print out a few reviews to make sure we mapped numpy array to plain text properly.

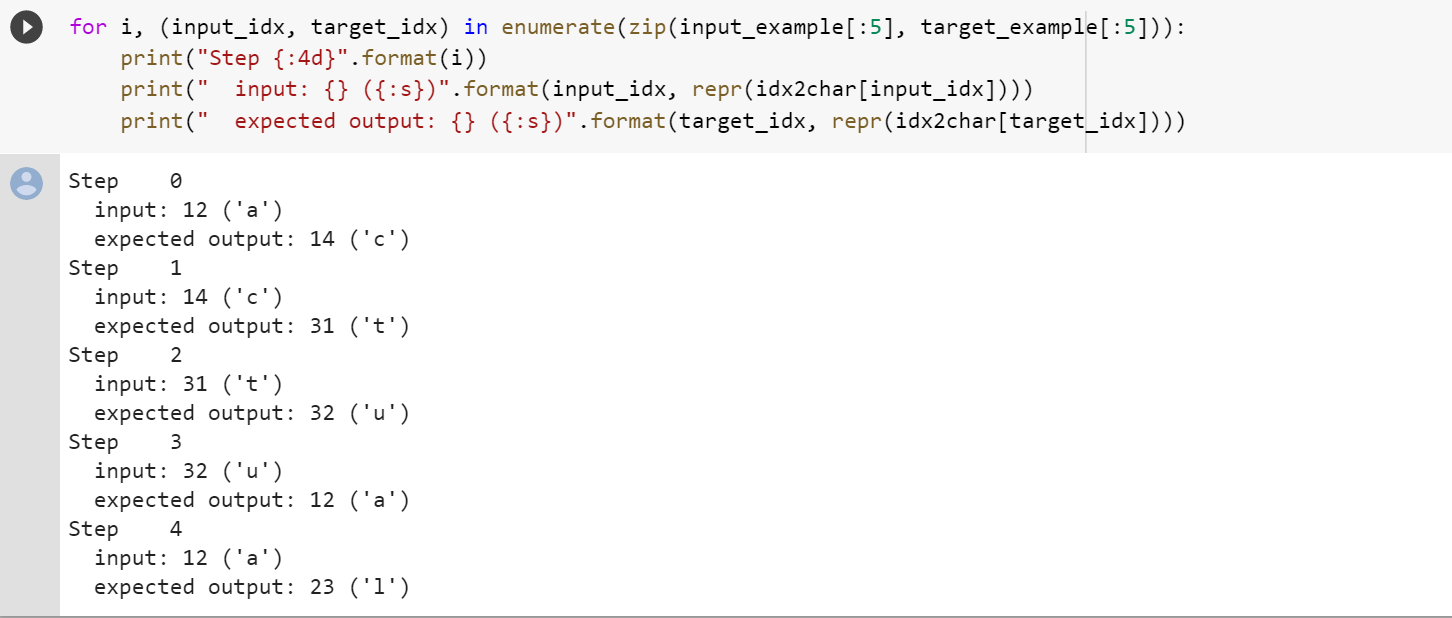


## **Input and target for text generation**

Now we need to train a text generation model. The model requires training data and targets. Conceptually, text generation implies that given an input, the model generates an output, character by character. So we may create target as a one-position offset of the input. split\_input\_target function below accomplishes this.

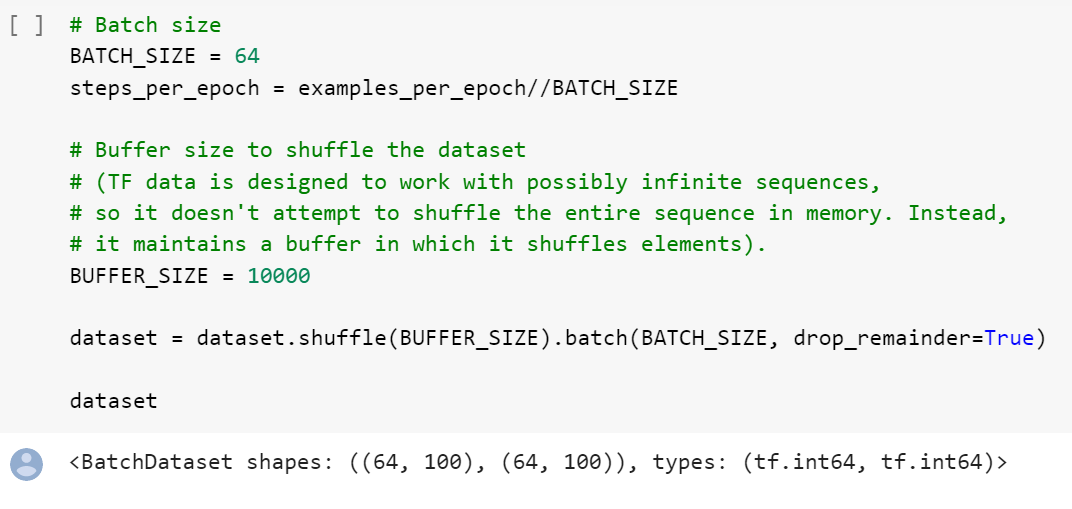


Each index of these vectors are processed as one time step. For the input at time step 0, the model receives the index for "F" and tries to predict the index for "i" as the next character. At the next timestep, it does the same thing but the RNN considers the previous step context in addition to the current input character.



Create training batches

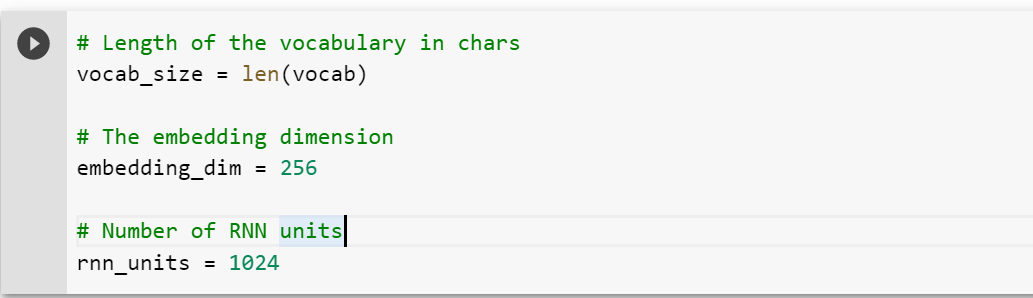
We used tf.data to split the text into manageable sequences. But before feeding this data into the model, we need to shuffle the data and pack it into batches.

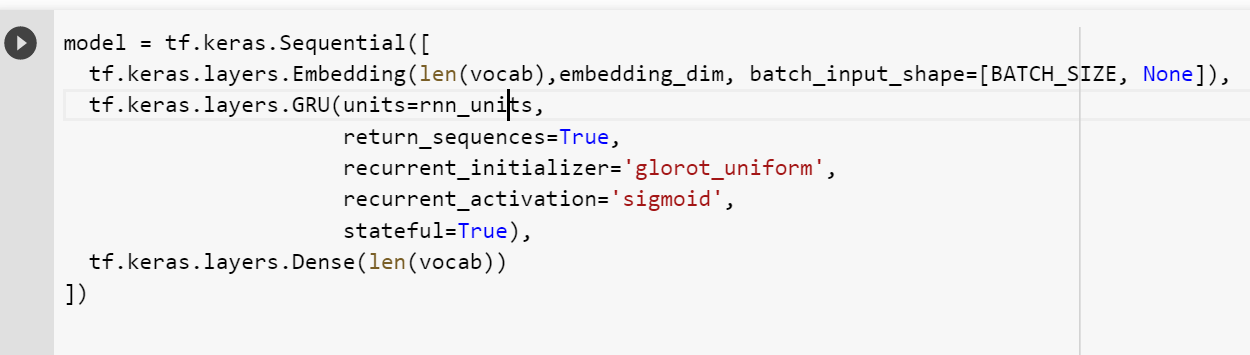


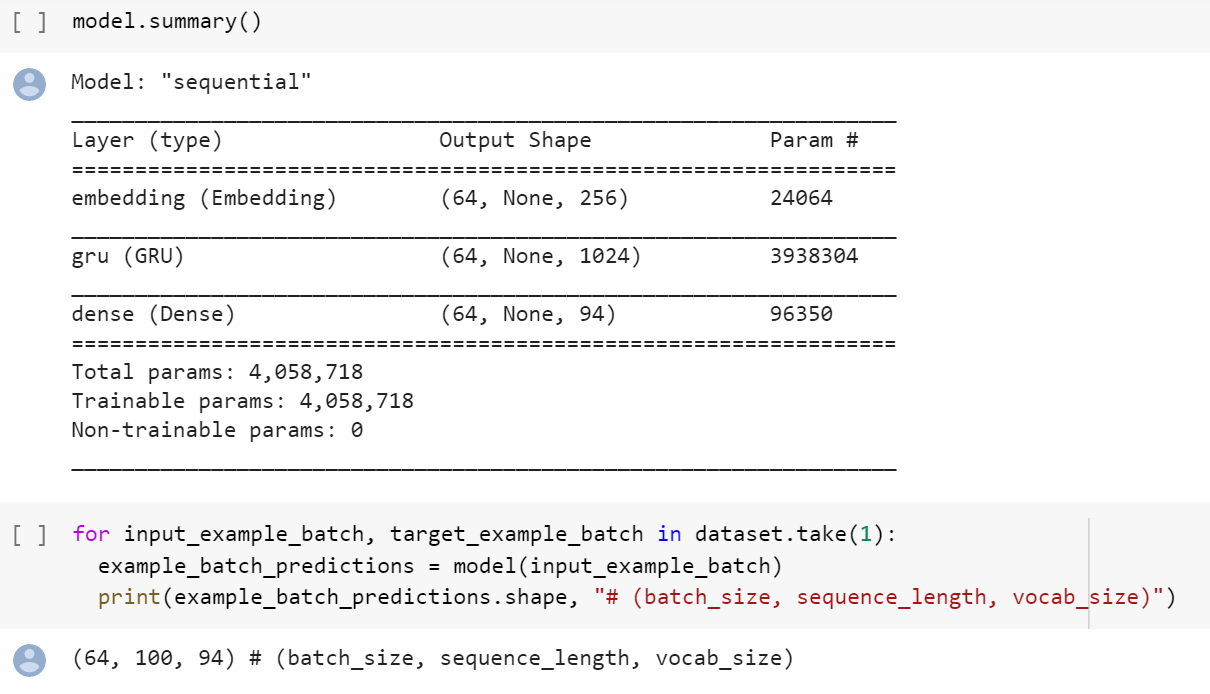
## **Build The Model**

Use tf.keras.Sequential to define the model. For this simple example three layers are used to define our model:

tf.keras.layers.Embedding: The input layer. A trainable lookup table that will map the numbers of each character to a vector with embedding\_dim dimensions; tf.keras.layers.GRU: A type of RNN with size units=rnn\_units (You can also use a LSTM layer here.) tf.keras.layers.Dense: The output layer, with vocab\_size outputs.





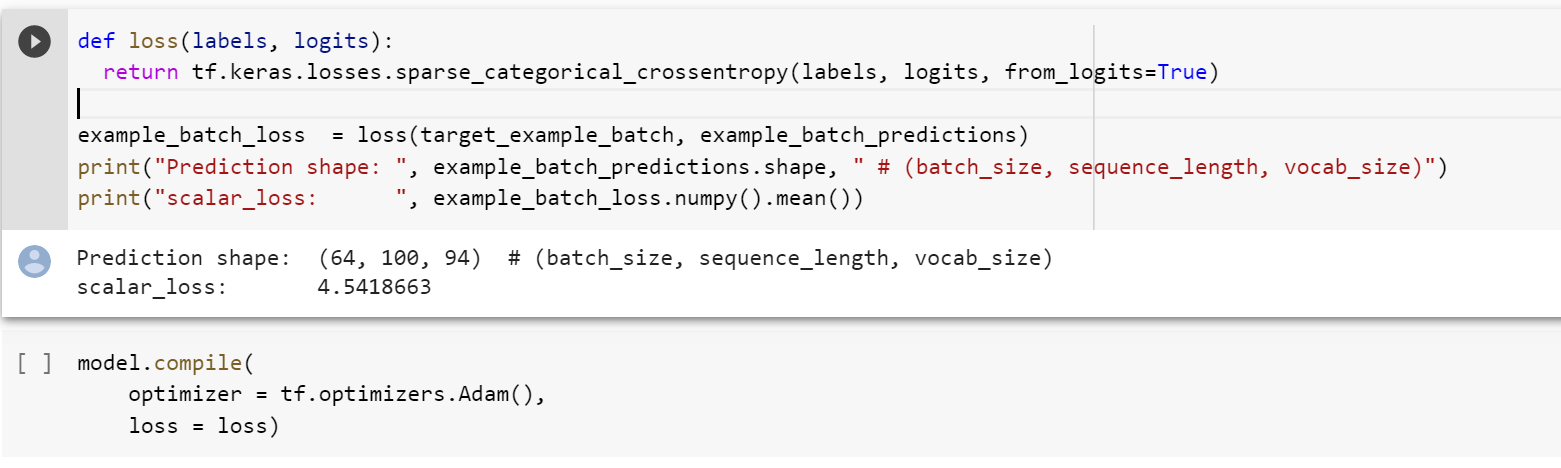


Next we need to define a loss function for the training. Then designate Adam optimizer for model compile.

### Attach an optimizer, and a loss function

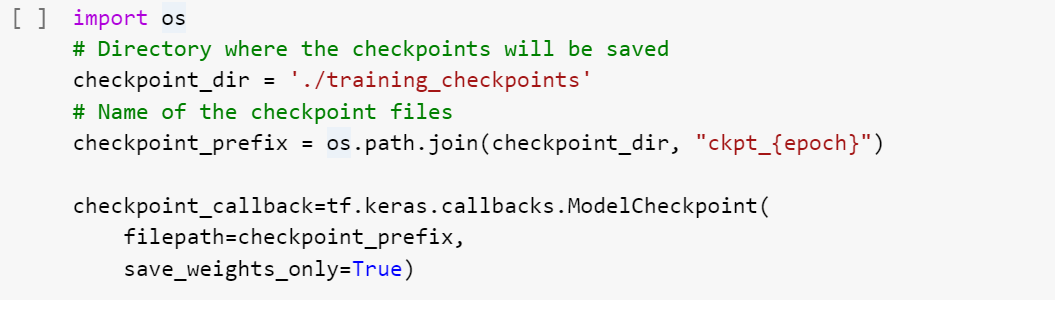
The standard tf.keras.losses.sparse\_softmax\_crossentropy loss function works in this case because it is applied across the last dimension of the predictions.

Because our model returns logits, we need to set the from\_logits flag.

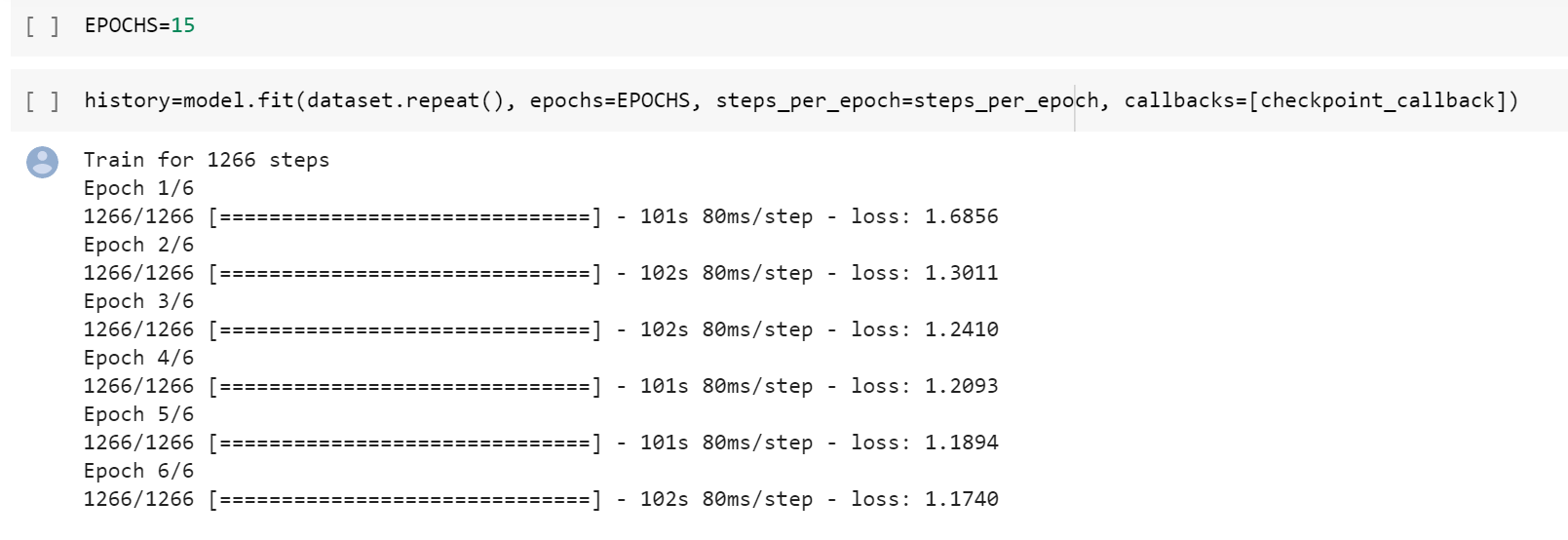


### Configure checkpoints

Use a tf.keras.callbacks.ModelCheckpoint to ensure that checkpoints are saved during training:



Now we are ready for training a text generation model. Feel free to adjust training epochs.



At this point the problem can be treated as a standard classification problem. Given the previous RNN state, and the input this time step, predict the class of the next character.

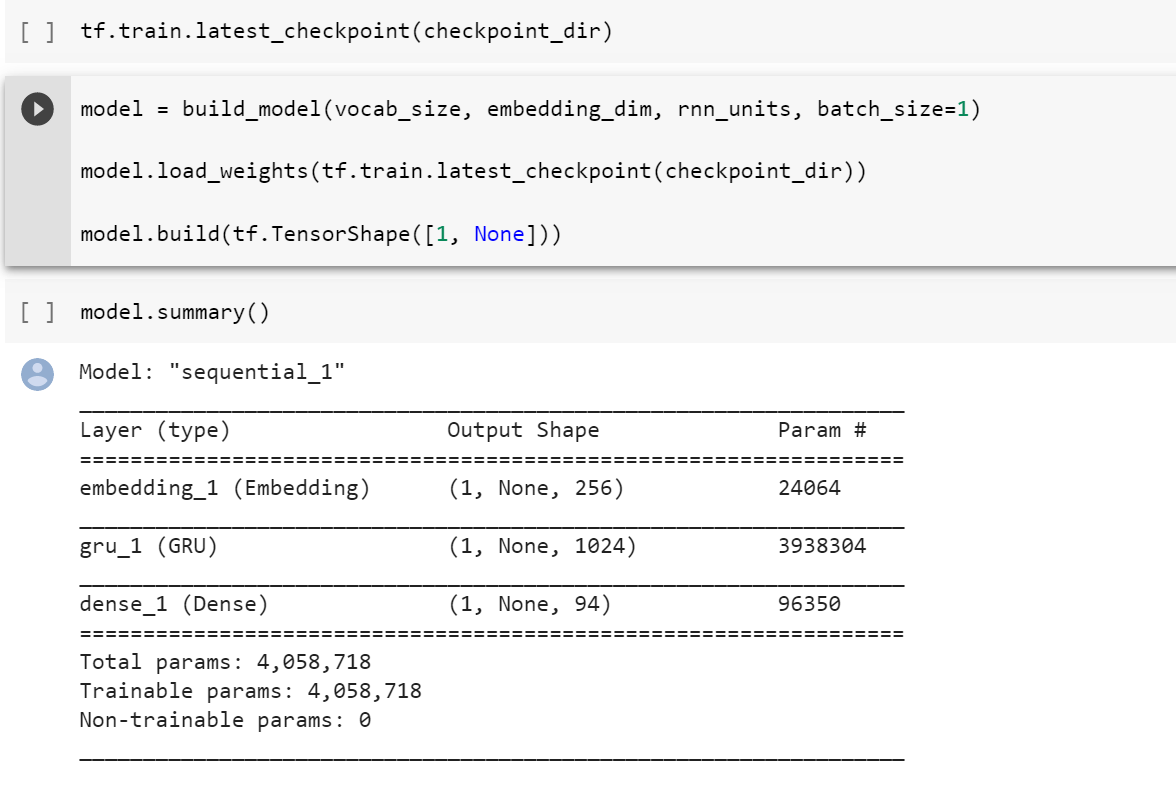
## **Generate text**

### Restore the latest checkpoint

To keep this prediction step simple, use a batch size of 1.

Because of the way the RNN state is passed from timestep to timestep, the model only accepts a fixed batch size once built.

To run the model with a different batch\_size, we need to rebuild the model and restore the weights from the checkpoint.



### The prediction loop

The following code block generates the text:

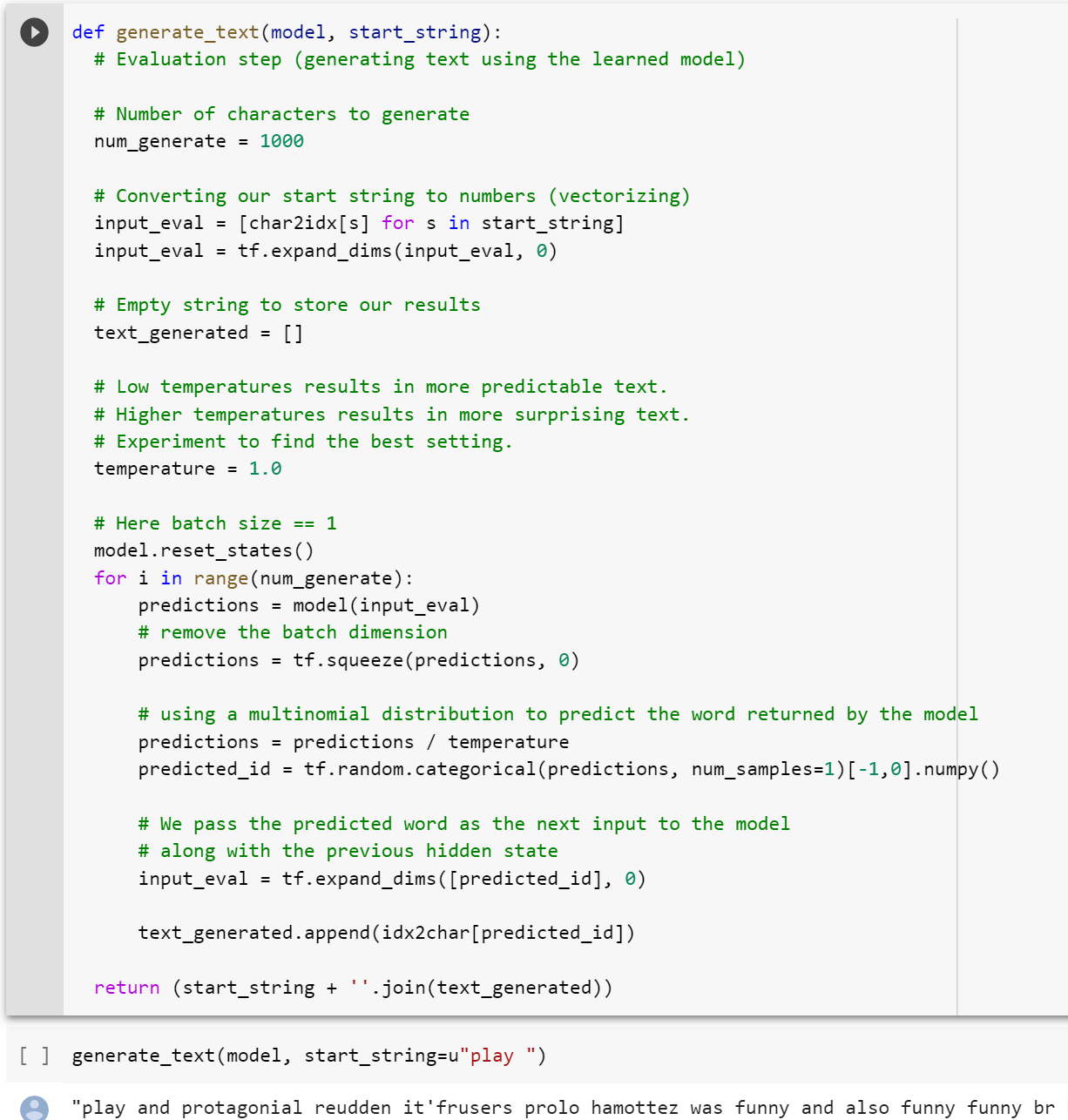
* It Starts by choosing a start string, initializing the RNN state and setting the number of characters to generate.
* Get the prediction distribution of the next character using the start string and the RNN state.
* Then, use a multinomial distribution to calculate the index of the predicted character. Use this predicted character as our next input to the model.
* The RNN state returned by the model is fed back into the model so that it now has more context, instead than only one word. After predicting the next word, the modified RNN states are again fed back into the model, which is how it learns as it gets more context from the previously predicted words.

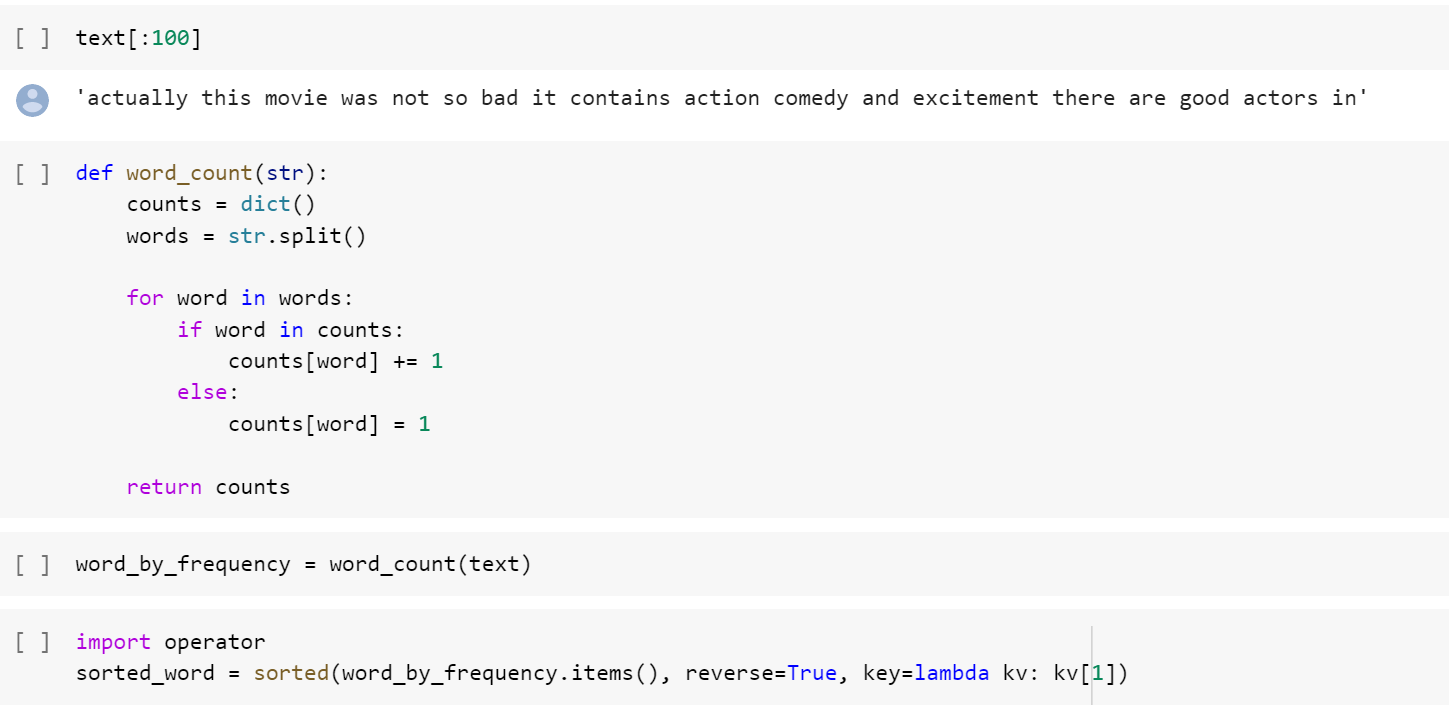
Looking at the generated text, you'll see the model knows and imitates vocabulary. With the small number of training epochs, it has not yet learned to form coherent sentences.

## **Generating Text**

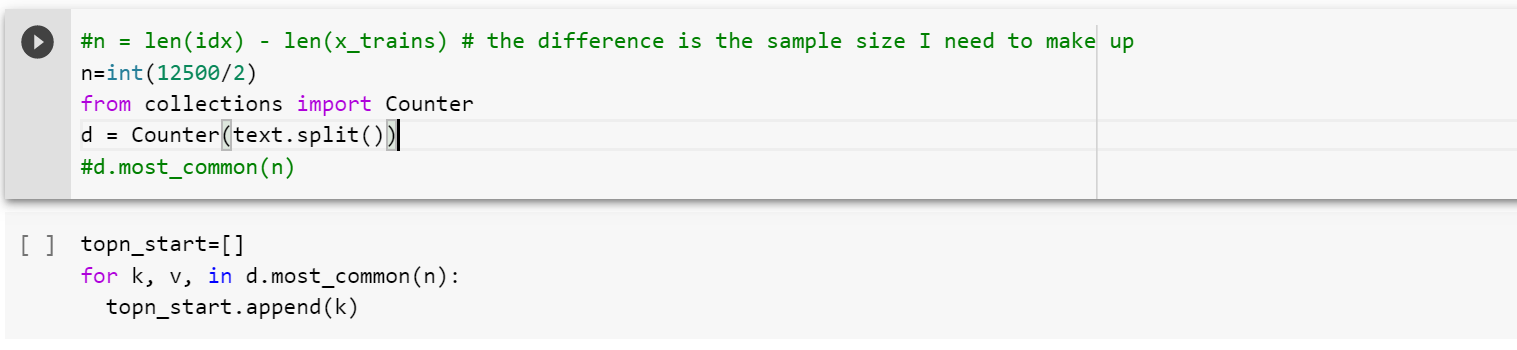
#### **Starting seed**

In order to generate text, we need to supply the model a starting 'seed'. Lets Rank unique words in the IMDB texts by their frequency in descending order, and select top n words as our seeds. Each seed will allow the model to generate one string of text until the specified maximum length.





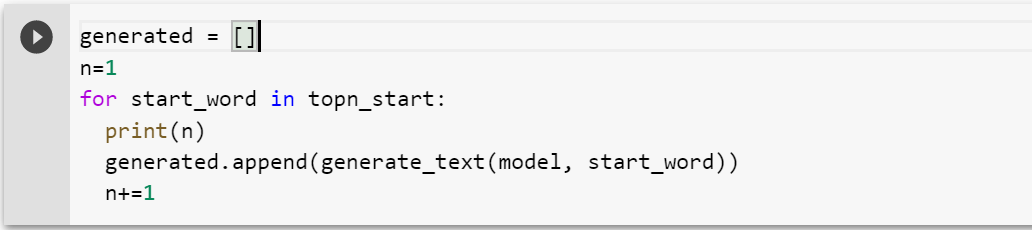
I need to have 12500 positive samples. Since I only have half to start with, I need 6250 extra new positive samples to be generated by my model. I want to get top 6250 words from dictionary by frequency as the starting seed.



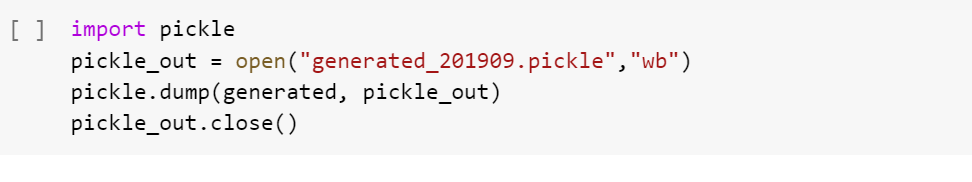
Lets examine one sample of generated positive review. See if it has the look or feels that resembles a real positive review



We are ready to generate new sentences, encode it to numpy array, merge with the original positive reviews.



Lets save this list for future use:



### Save generated text for convenience

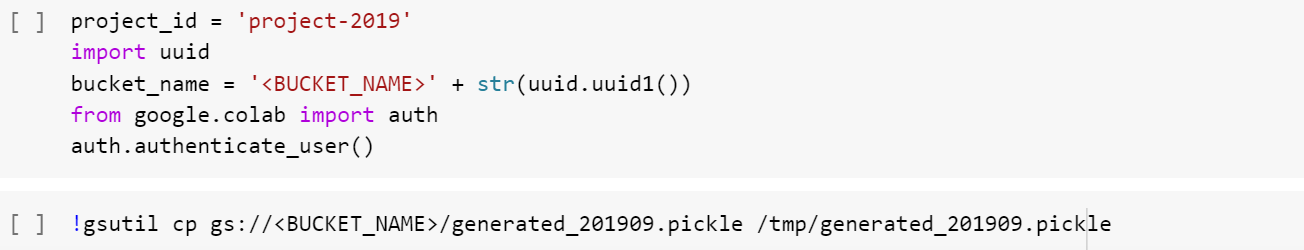
We may persist generated reviews as a pickle file to a cloud resource such as Google Cloud Platform (GCP) storage bucket. Steps below requires you to have an active GCP account.

If you do not have a GCP account or simply wishes to continue using example generated reviews provided by the course author, please skip the steps below and go to the next section "Load generated positive reviews provided by author"



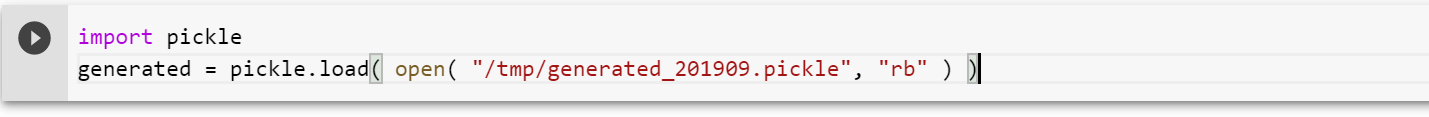
### Load generated positive reviews provided by author

Lets set up authentication for accessing GCP bucket, where the generated positive reviews is stored by the author.

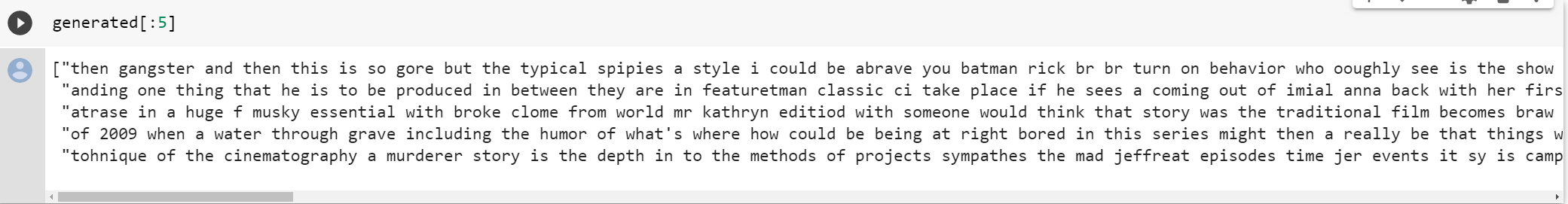


Load generated positive reviews generated and provided by the author. It is a pickle file.

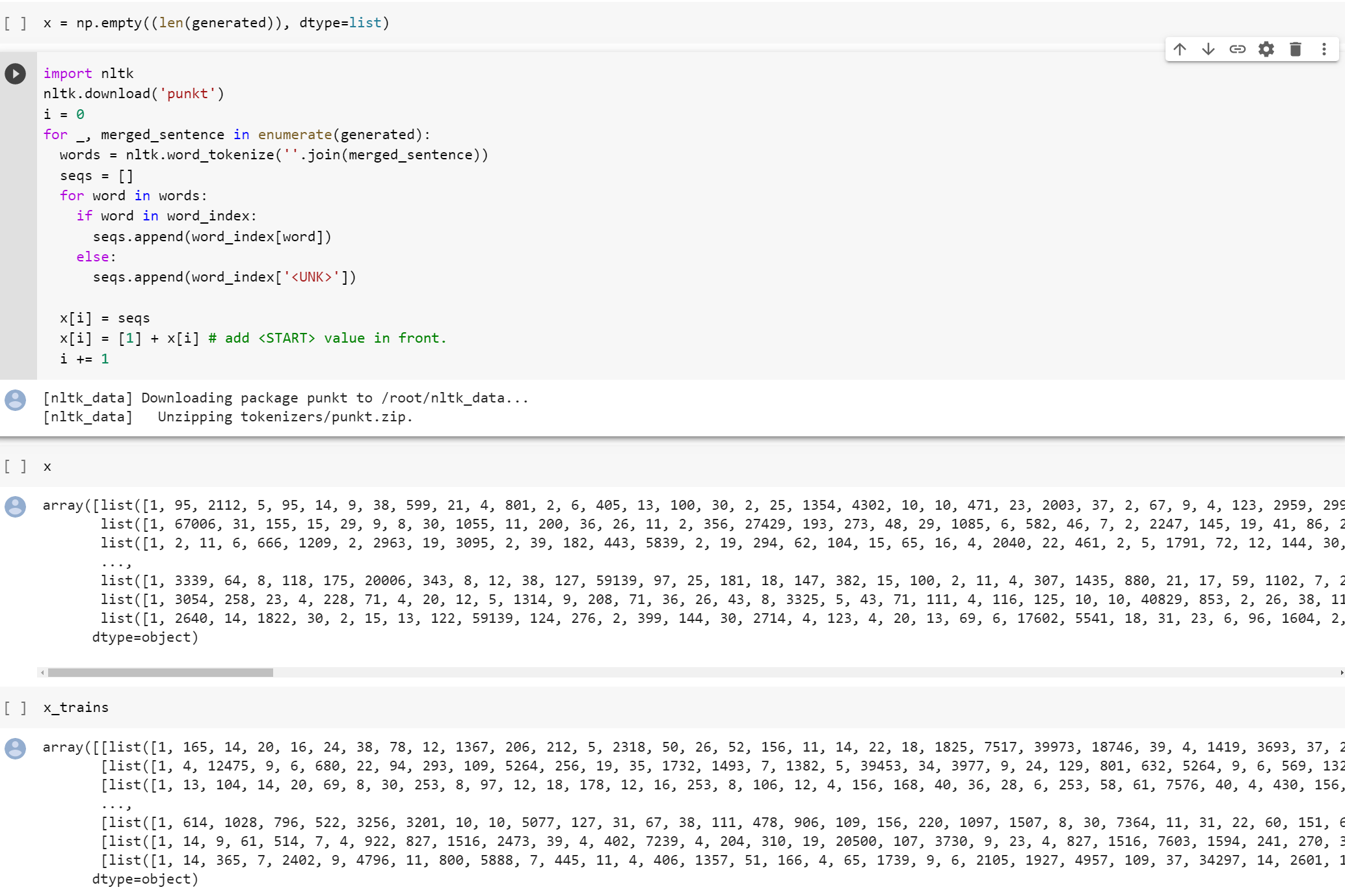
The code cell below assumes the pickle file is put in /tmp directory relative to the current working directory. Your situation may vary. You may specify full file path if you wish.



Take a look at content



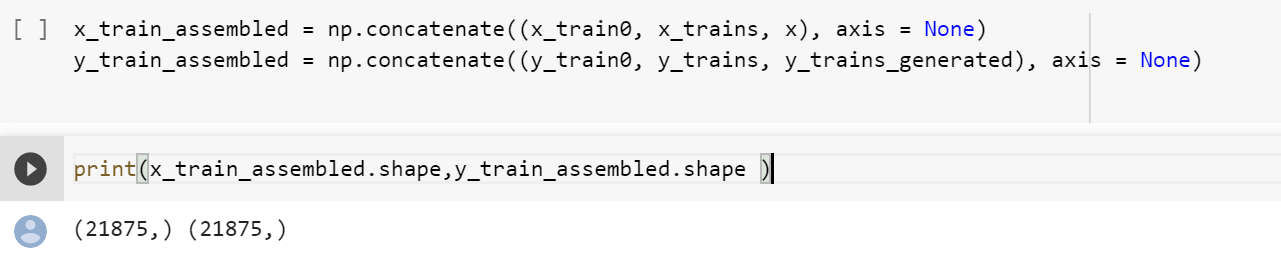
Lets now tokenize the generated reviews using Punkt tokenizer provided in the NLTK library.



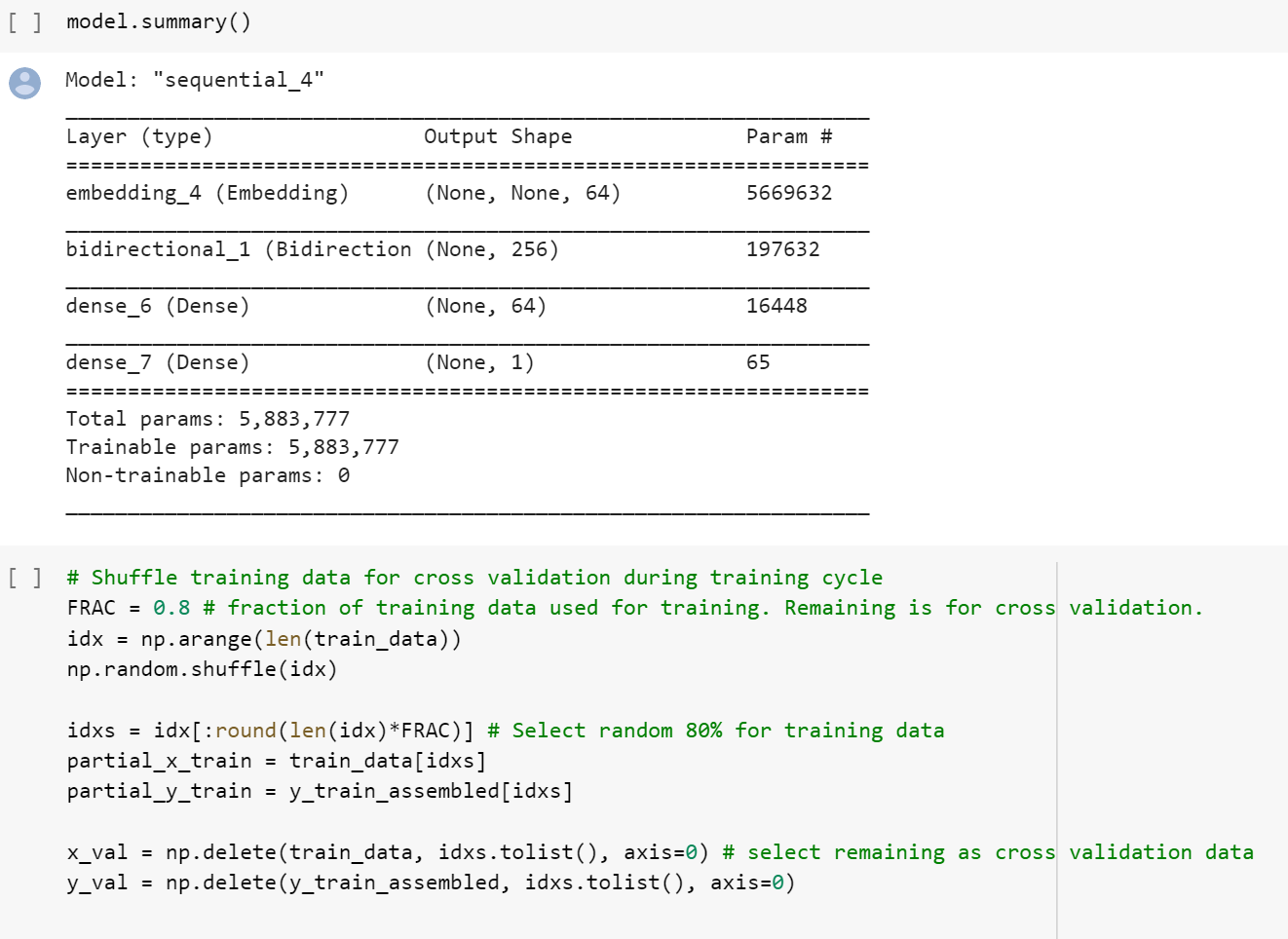
We also need to create label for these generated text as positive review. Lets just use the training label already provided by the original data.

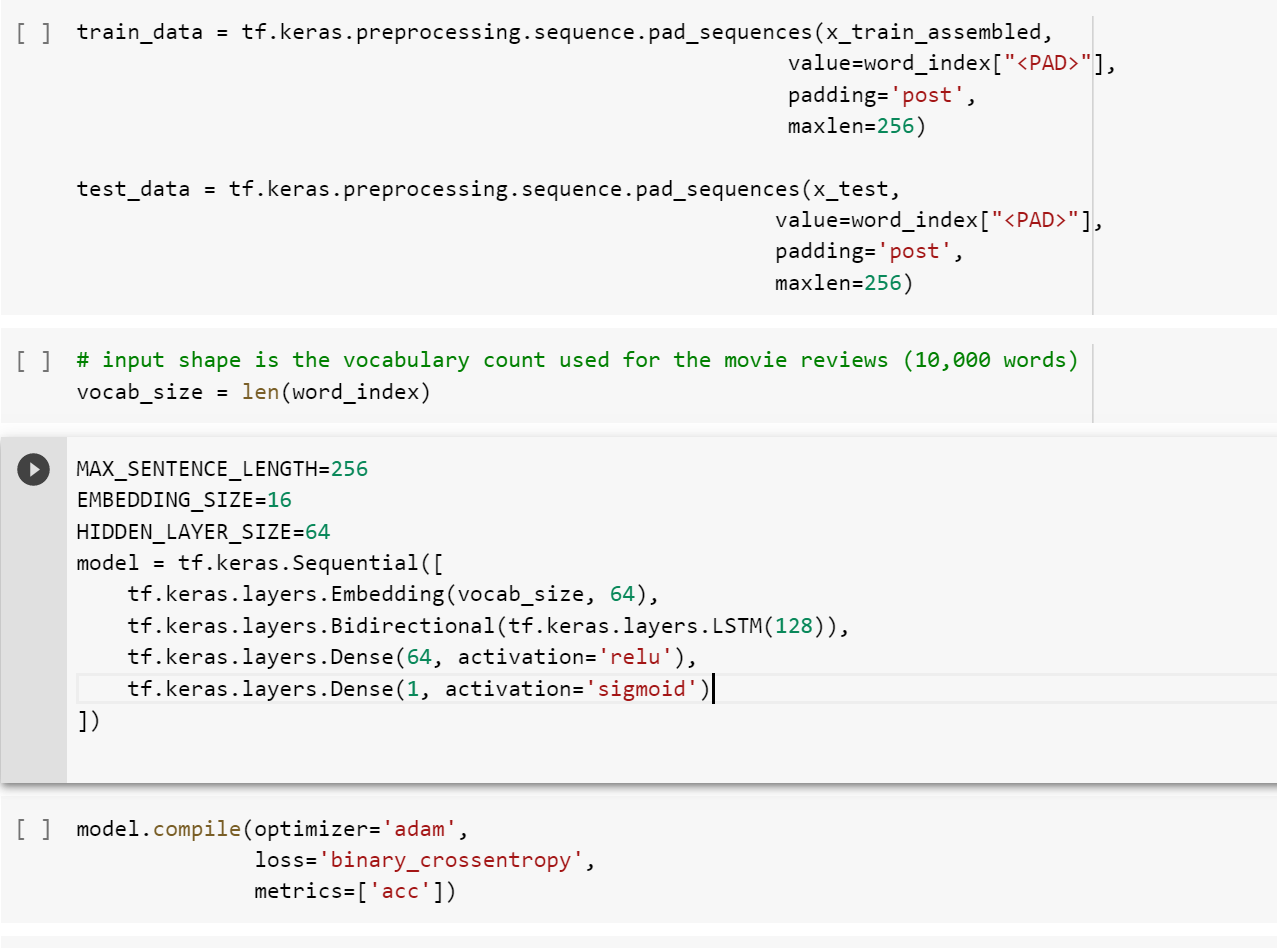


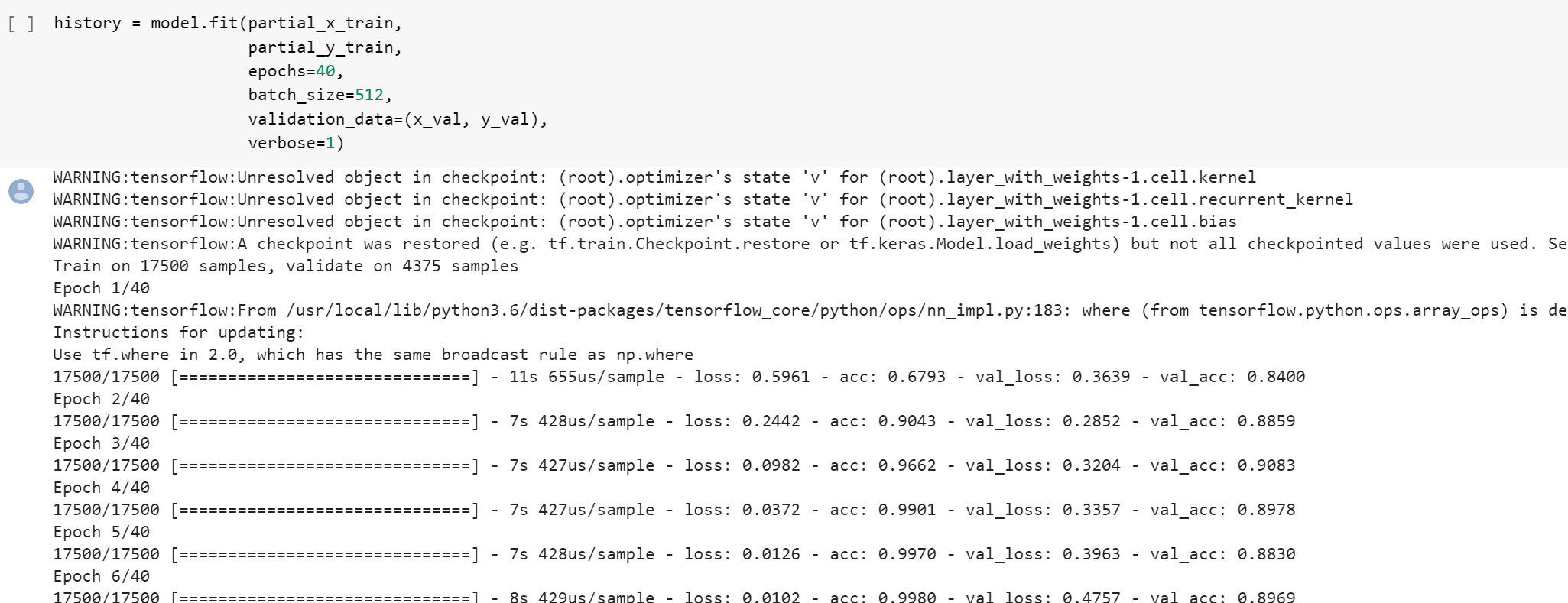
Now we can put together original reviews with the generated reviews, and likewise their respective labels.



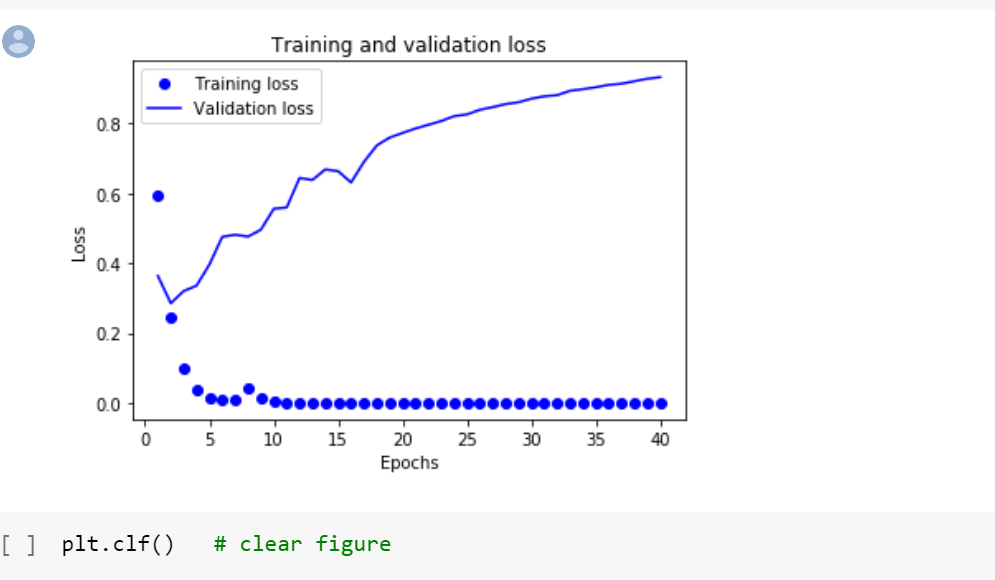
Take care of padding.



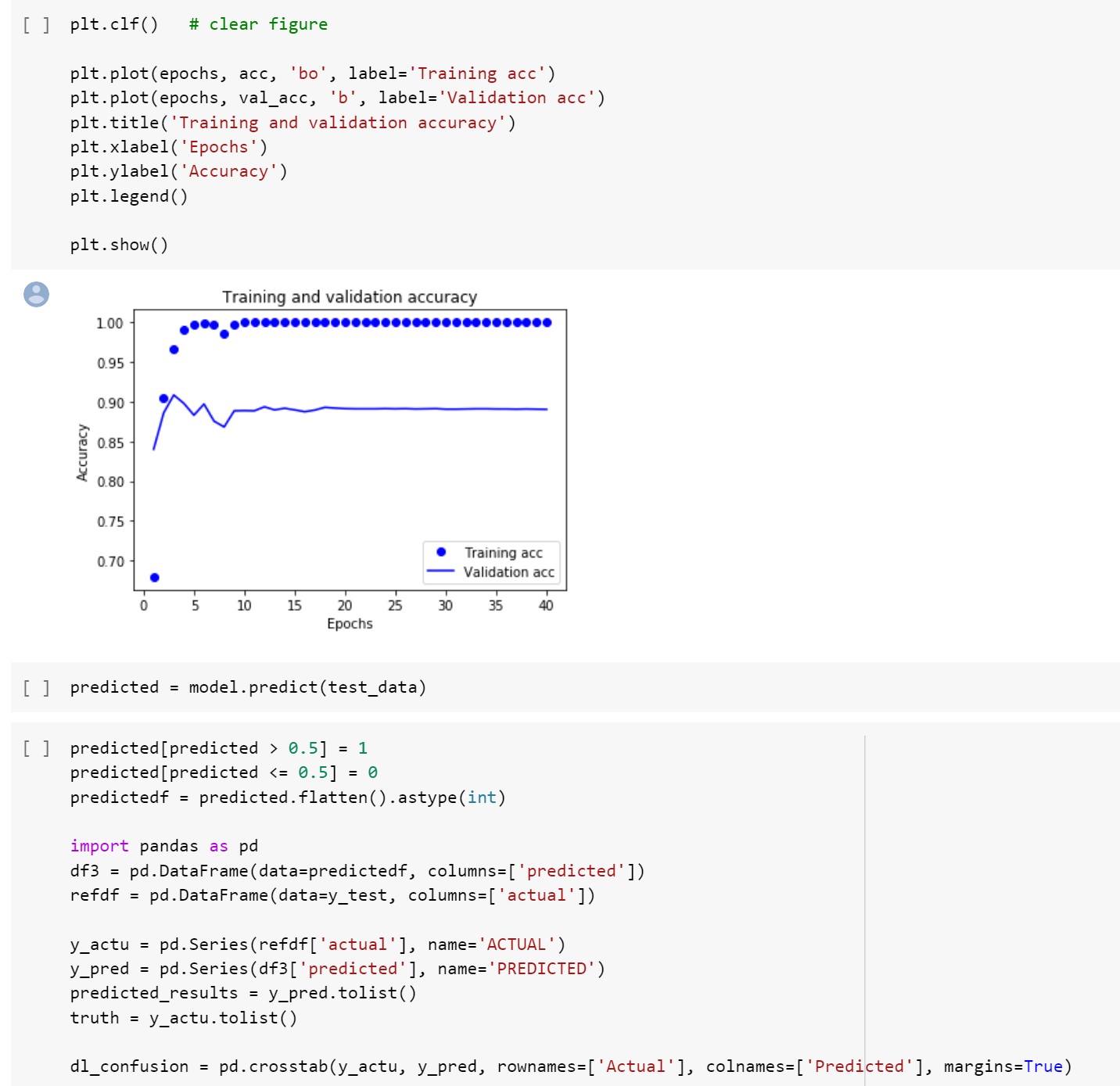


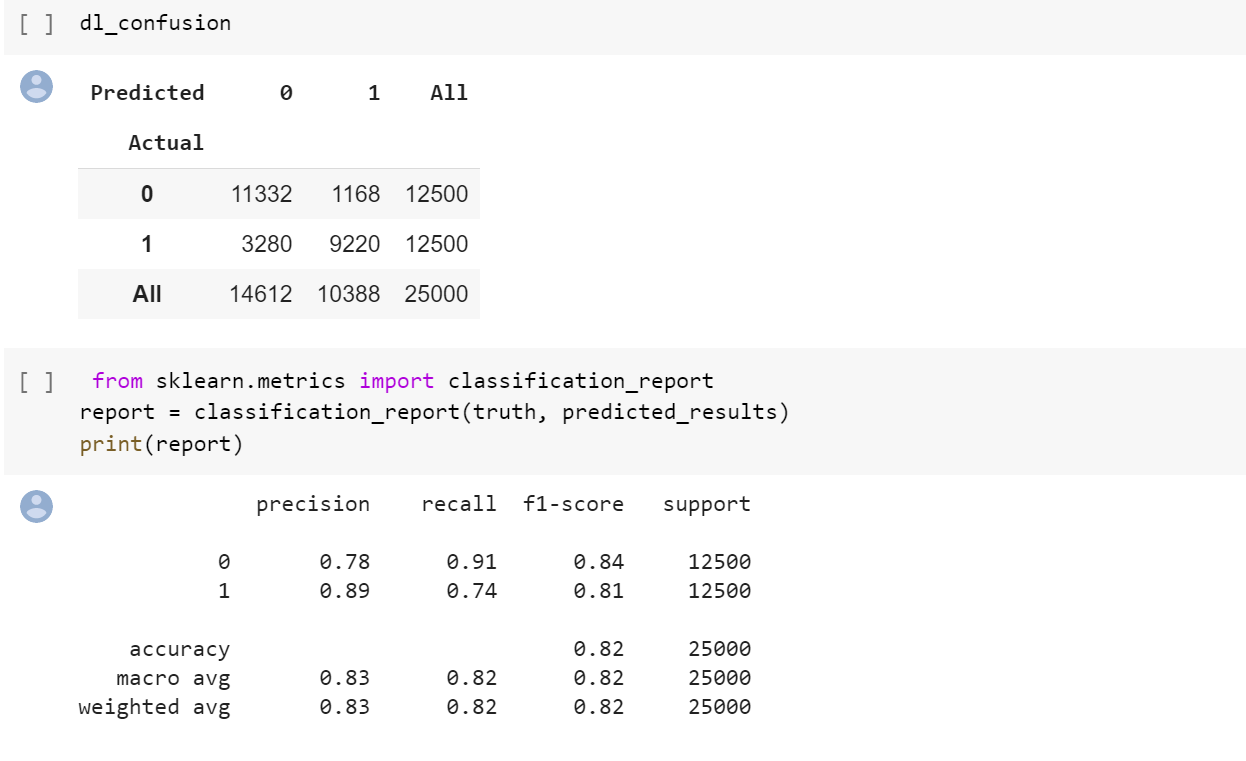






Now lets take a look at training progression and create a confusion matrix for testing results.





Model produces reasonable precision and recall for positive reviews classification. This shows that machine text generation can produce realistic texts to make up for imbalance text dataset.